Módulo I – Fundamentos Computacionais
Sessão 5 - Aula 6 - Recursão

Prof. Dr. Luiz Alberto Vieira Dias
Prof. Dr. Lineu Mialaret

Recursão
Introdução

- Um modo de se descrever tarefas repetitivas em um programa é por meio das instruções de laços (*loops*)
  - No Python, tem-se as instruções *while* e *for*
- Um modo diferente de se implementar repetição é por meio de recursão
- Recursão é uma técnica pela qual
  - Uma função faz uma ou mais chamadas para si mesma durante a execução do programa
  - Uma estrutura de dados depende de instâncias menores do mesmo tipo de estrutura na sua representação

Recursão
Introdução (cont.)

- Exemplos de recursão:

Recursão
Fatorial

- O fatorial de um inteiro positivo ***n***, designado por ***n*!**, é definido como o produto de inteiros de 1 a ***n***
- Se ***n*** = 0, então ***n*!** é definido como 0 por convenção
- Formalmente

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

- Exemplo:
  - 5! = 5.4.3.2.1 = 120
  - 20! = ?

Recursão
Fatorial (cont.)

- Importância do fatorial
  - É o número de modos pelos quais ***n*** itens distintos podem ser montados em uma sequência (o número de permutações de uma sequência)
- Exemplo:
  - Dado três caracteres *a*, *b* e *c*, eles podem ser arranjados em 3! = 3.2.1 = 6 modos
    - *abc*, *acb*, *bac*, *bca*, *cab*, *cba*

Recursão
Fatorial (cont.)

- Há uma definição recursiva para a função fatorial, que pode ser formalizada como se segue

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

Caso base

Caso recursivo

- A definição acima contém
  - Um caso base, que é definido em termos de um valor fixo
  - Um caso recursivo, definido pela utilização da própria definição da função que está sendo definida

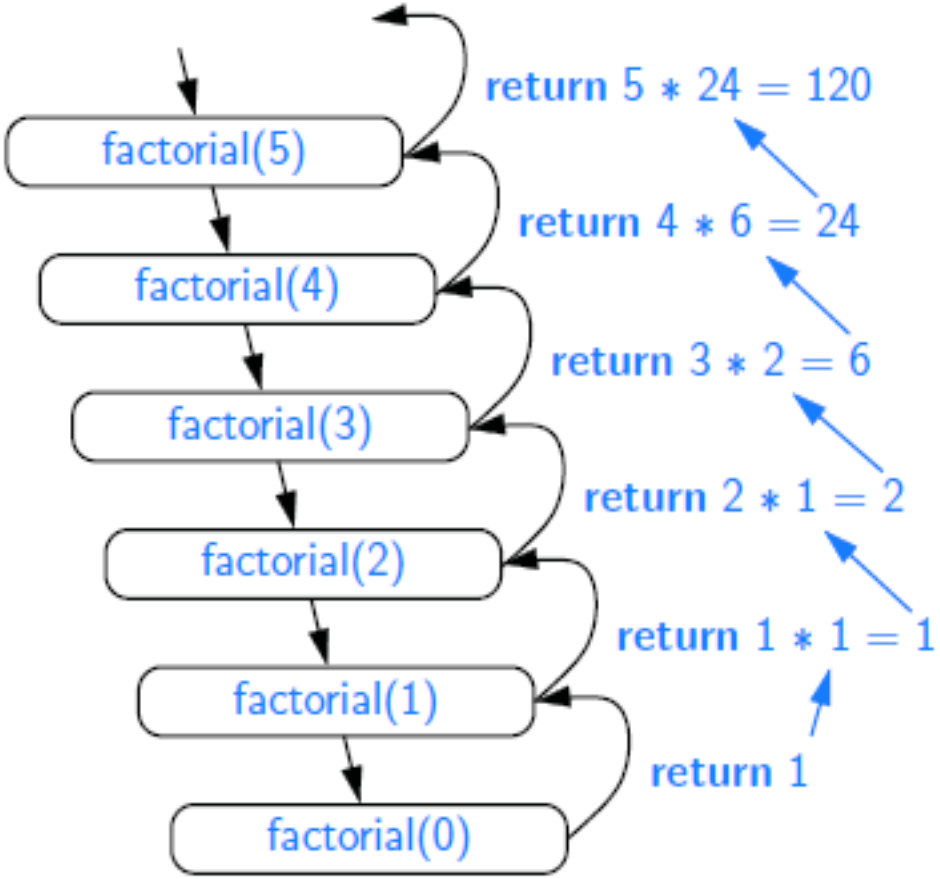
Recursão
Fatorial – Implementação em Python

- Características do código:
  - Não tem laços explícitos
  - Repetição acontece por chamadas recursivas repetidas da função
  - Cada vez que a função é invocada, seu argumento é diminuído de uma unidade
  - No caso base, nenhuma chamada é feita

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n-1)
```

Recursão
Fatorial (cont.)

- Pode-se ilustrar a execução de uma função recursiva utilizando-se o rastreamento de recursão (*recursion trace*)

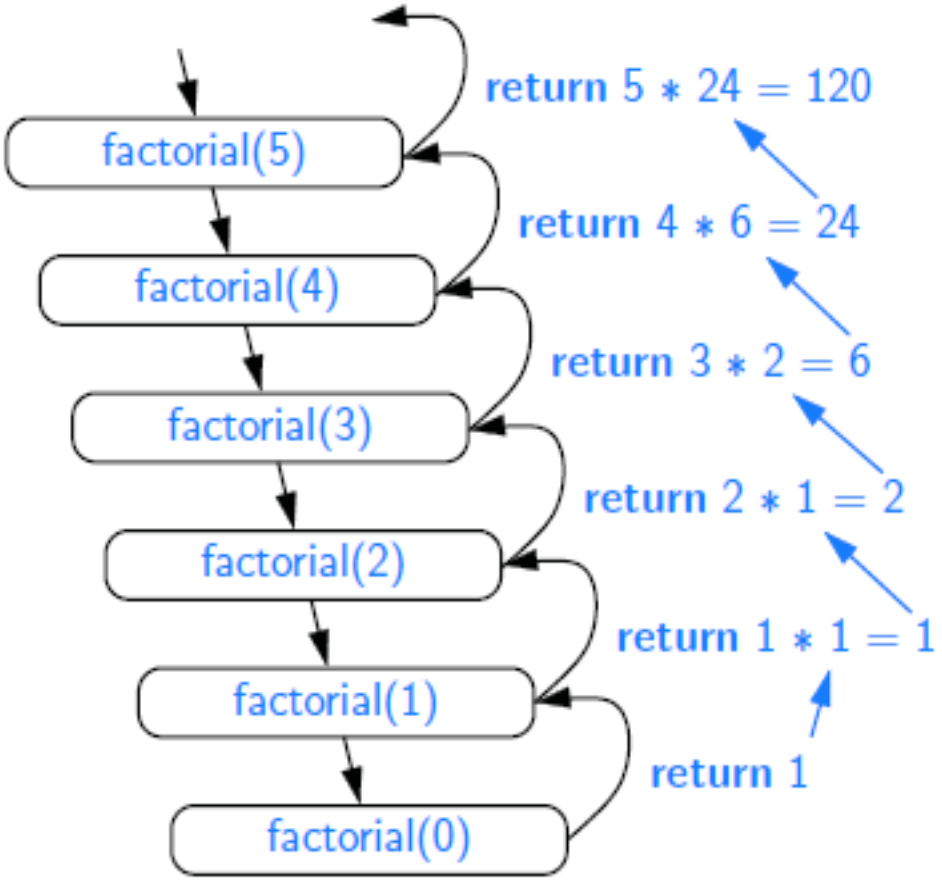


- Uma caixa para cada chamada recursiva
- Um seta de cada chamador para a função invocada
- Uma seta de cada função invocada mostrando o retorno

Recursão
Fatorial (cont.)

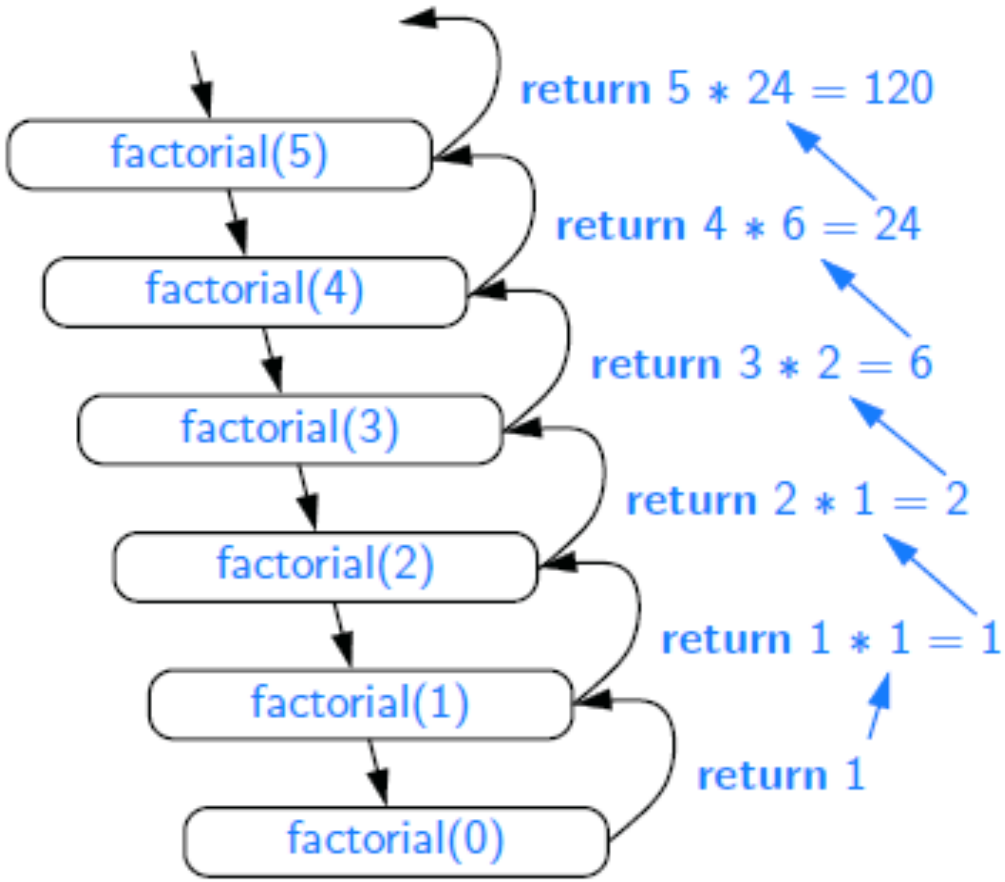
- Pode-se ilustrar a execução de uma função recursiva utilizando-se o rastreamento de recursão (*recursion trace*)

```
def factorial(n):
  print ("activation {0}".format(n))
  if n == 0:
    return 1
  else:
    return n * factorial(n-1)
fact = factorial(5)
print (fact)
```



Recursão
Fatorial (cont.)

- Pode-se ilustrar a execução de uma função recursiva utilizando-se o rastreamento de recursão (*recursion trace*)



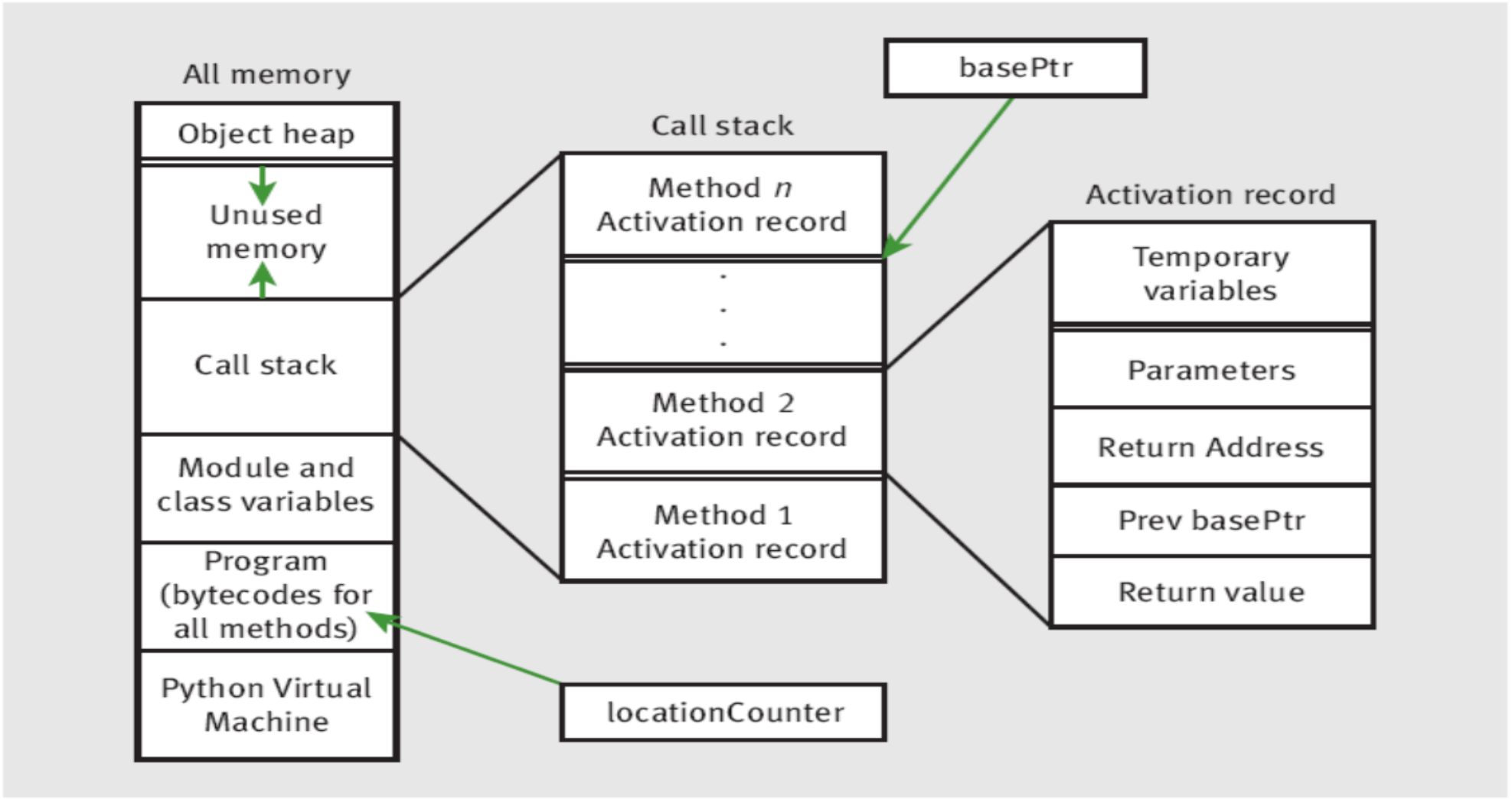

```
def factorial(n):
  print ("activation {0}".format(n))
  if n == 0:
    return 1
  else:
    return n * factorial(n-1)
fact = factorial(5)
print (fact)
```

```
activation 5
activation 4
activation 3
activation 2
activation 1
activation 0
120
```

Recursão
Registro de Ativação

- No Python, cada vez que uma função é invocada, uma estrutura denominada de registro de ativação (*activation record*) é criada para armazenar informação sobre a invocação da função
  - Esta estrutura armazena o nome de espaços, parâmetros de chamada, variáveis, etc.
- Na invocação de uma função, a execução do chamador é suspensa e seu registro de ativação armazena o local onde a execução deve continuar após o termino da chamada da função

Recursão
Registro de Ativação (cont.)



The architecture of a run-time environment

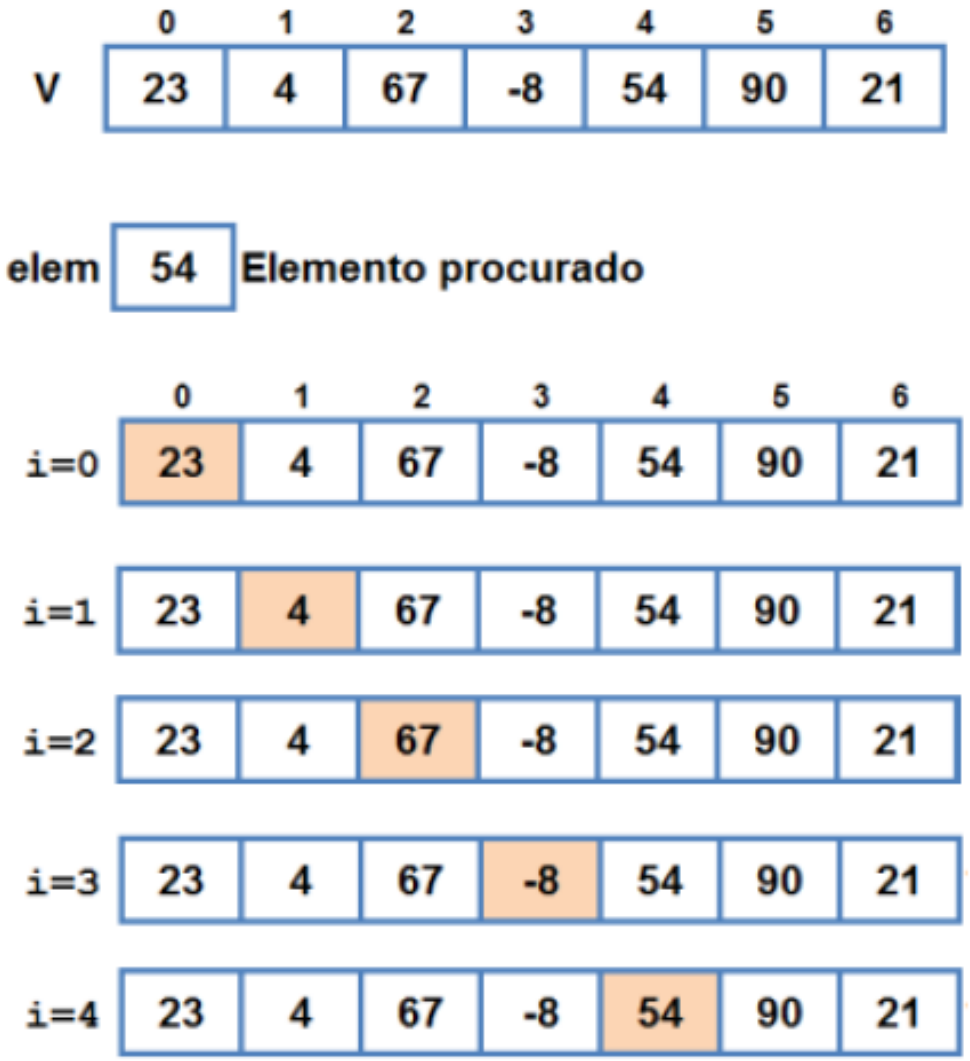
Recursão
Busca Binária

- O algoritmo de busca binária é um algoritmo clássico de recursão, utilizado para localizar um valor dentro de uma sequência ordenada de valores de ***n*** elementos
  - É um dos algoritmos mais importantes
  - É a razão de se frequentemente armazenar dados ordenados em uma sequência

Values stored in sorted order within an indexable sequence, such as a Python list. The numbers at top are the indices.

Recursão
Busca Binária (cont.)

- Quando a sequência de valores não está ordenada, utiliza-se um laço para examinar toda a sequência até encontrar o valor desejado ou se examinar todos os elementos
  - É o algoritmo de busca sequencial (*sequential search algorithm*)
  - Ele tem um tempo de execução de ***O***(***n***) no pior caso



Recursão
Busca Binária (cont.)

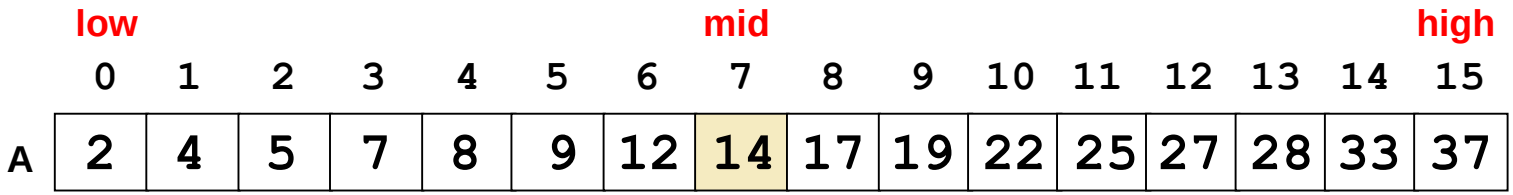
- Quando a sequência *A* está ordenada, indexada e tem tamanho ***n***, há um algoritmo mais eficiente
  - Para qualquer índice ***j*** da sequência de dados, sabe-se que todos os valores armazenados nos índices anteriores, 0,…,***j*** -1 são ≤ que os valores armazenados no índice ***j***
  - Sabe-se também que os valores armazenados nos índices posteriores, *j* +1,*j* +2,…,*n* -1 são ≥ que os valores armazenados no índice ***j***
  - Chama-se um elemento da sequência de candidato se, no estágio atual da pesquisa, não se pode dizer que ele é igual ao valor pesquisado

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Recursão
Busca Binária (cont.)

- O algoritmo mantém dois parâmetros, **low** e **high**, de modo que todos elementos candidatos tem um índice no mínimo **low** e no máximo **high**
- No inicio, **low** = 0 e **high** = ***n*** - 1
- Compara-se então o valor procurado com o elemento candidato médio, isto é, o elemento com índice

$$\text{mid} = \lfloor (\text{low} + \text{high})/2 \rfloor .$$

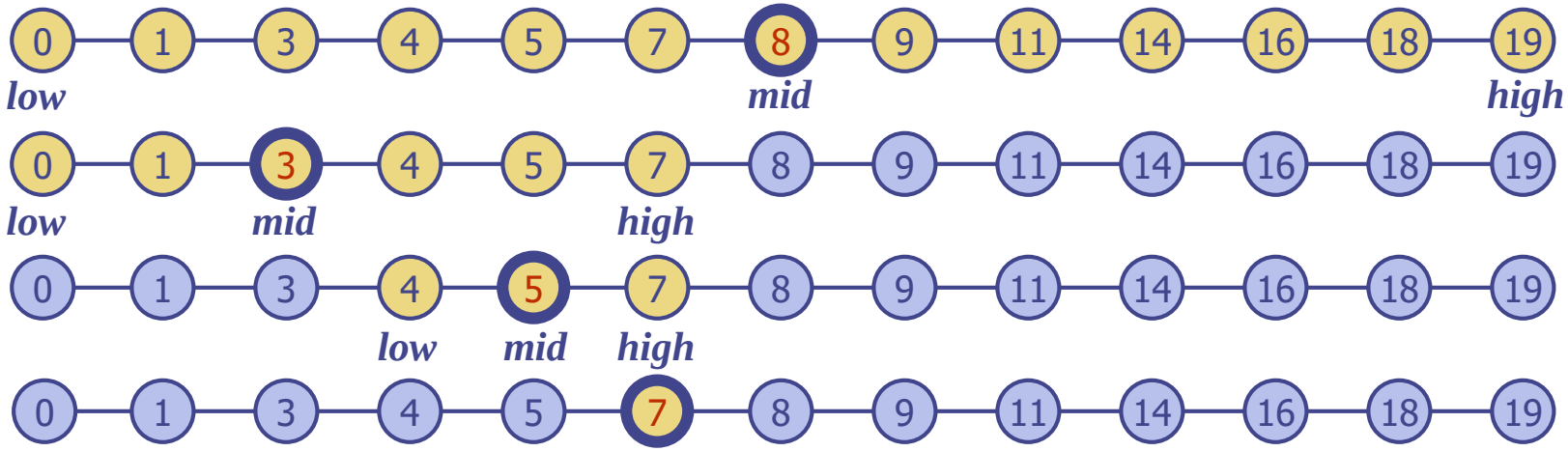


Recursão
Busca Binária (cont.)

- Pode-se considerar três casos:
  - Se o valor pesquisado é igual a **A[mid]**, então achou-se o item pesquisado e a pesquisa termina com sucesso
  - Se o valor pesquisado é < que **A[mid]**, então recursivamente vota-se a pesquisar na primeira metade da sequência, no intervalo de índices de **low** e **mid – 1**
  - Se o valor pesquisado é > que **A[mid]**, então recursivamente vota-se a pesquisar na segunda metade da sequência, no intervalo de índices de **mid +1** e **high**
- Uma busca sem sucesso ocorre se **low > high**, com o intervalo [low, high] vazio

Recursão
Busca Binária (cont.)

- Exemplo: Busca pelo valor 7



Recursão
Busca Binária (cont.)

- Implementação em Python do algoritmo de busca binária

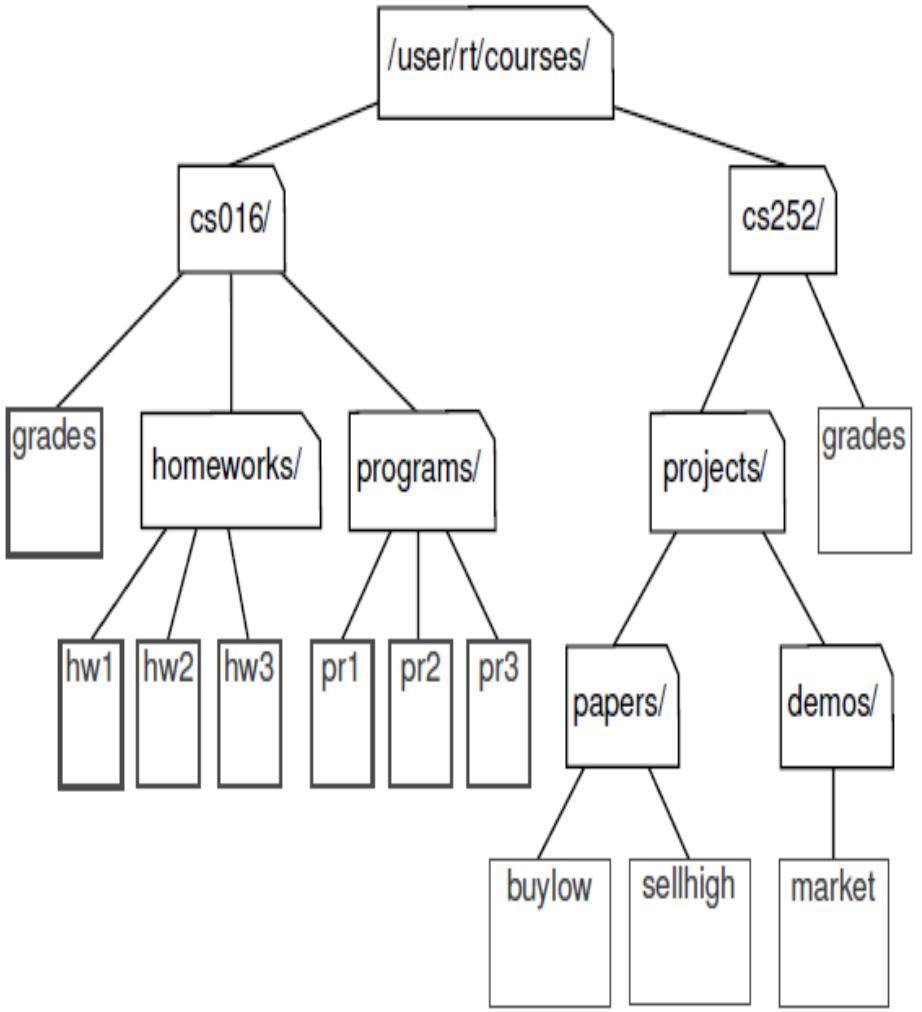
```
def binary_search(data, target, low, high):
  """Return True if target is found in indicated portion of a Python
list.
The search only considers the portion from data[low] to data[high]
inclusive.
  """
  if low > high:
    return False                        # interval is empty; no match
  else:
    mid = (low + high) // 2
    if target == data[mid]:             # found a match
      return True
    elif target < data[mid]:
      # recur on the portion left of the middle
      return binary_search(data, target, low, mid - 1)
    else:
      # recur on the portion right of the middle
      return binary_search(data, target, mid + 1, high)
```

Recursão
Busca Binária (cont.)

- Considerações da implementação em Python do algoritmo de busca binária
  - O algoritmo de busca sequencial tem um tempo de execução de **O(*n*)**
  - O algoritmo de busca binária tem um tempo de execução **O(log *n*)**
- O algoritmo de busca binária representa um melhoramento significativo, pois se o tamanho da sequência é ***n*** = 1.000.000.000, log ***n*** = 30

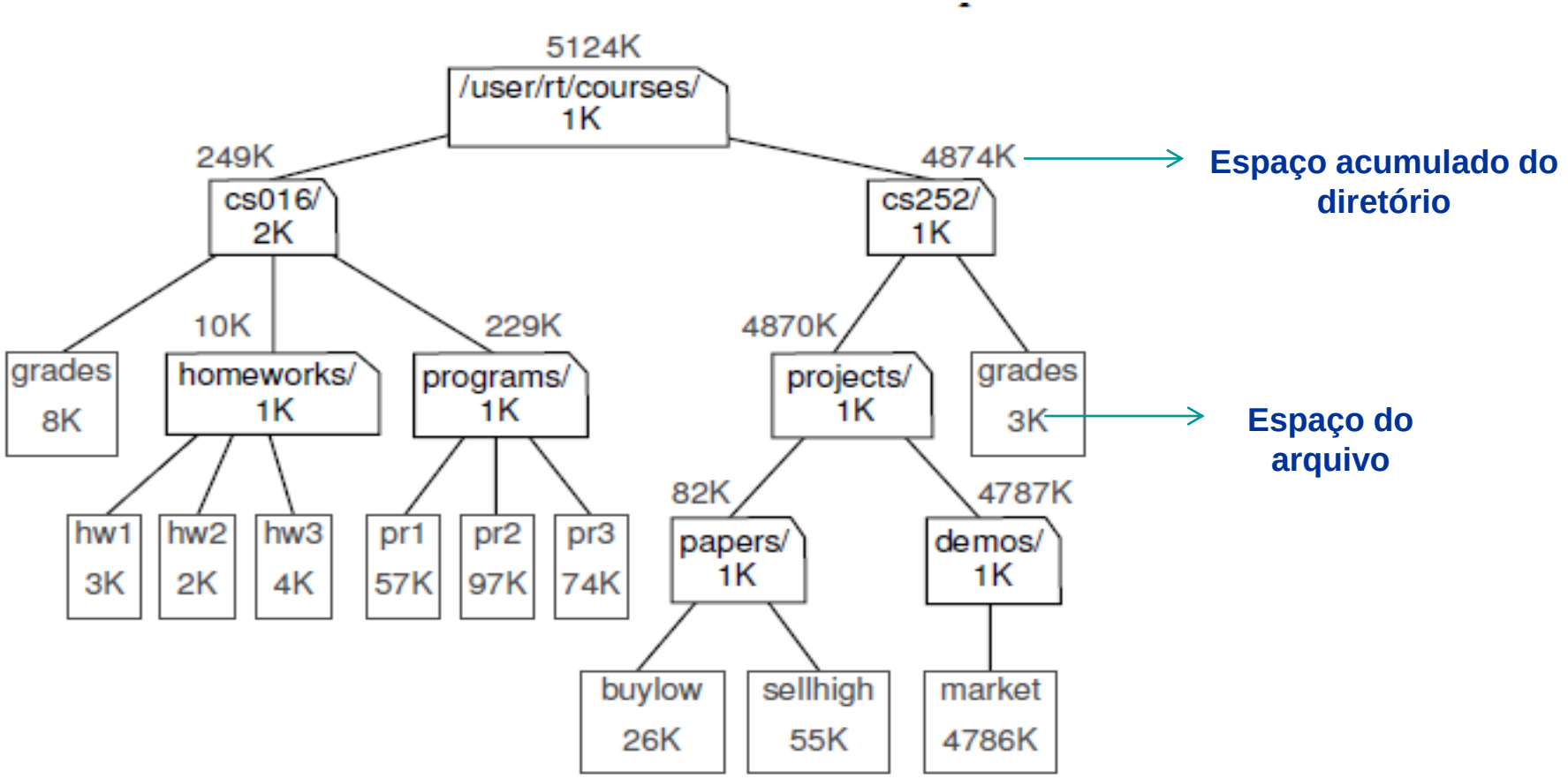
Recursão
Arquivos de Sistemas

- Sistemas operacionais modernos definem uma estrutura de armazenamento chamada diretórios de arquivos de sistemas (*file-system directories* ou *folders*)
  - Um arquivo de sistema consiste de um diretório de alto nível e o conteúdo deste diretório consiste de arquivos e outros diretórios, que por seu turno podem conter arquivos e diretórios e assim por diante



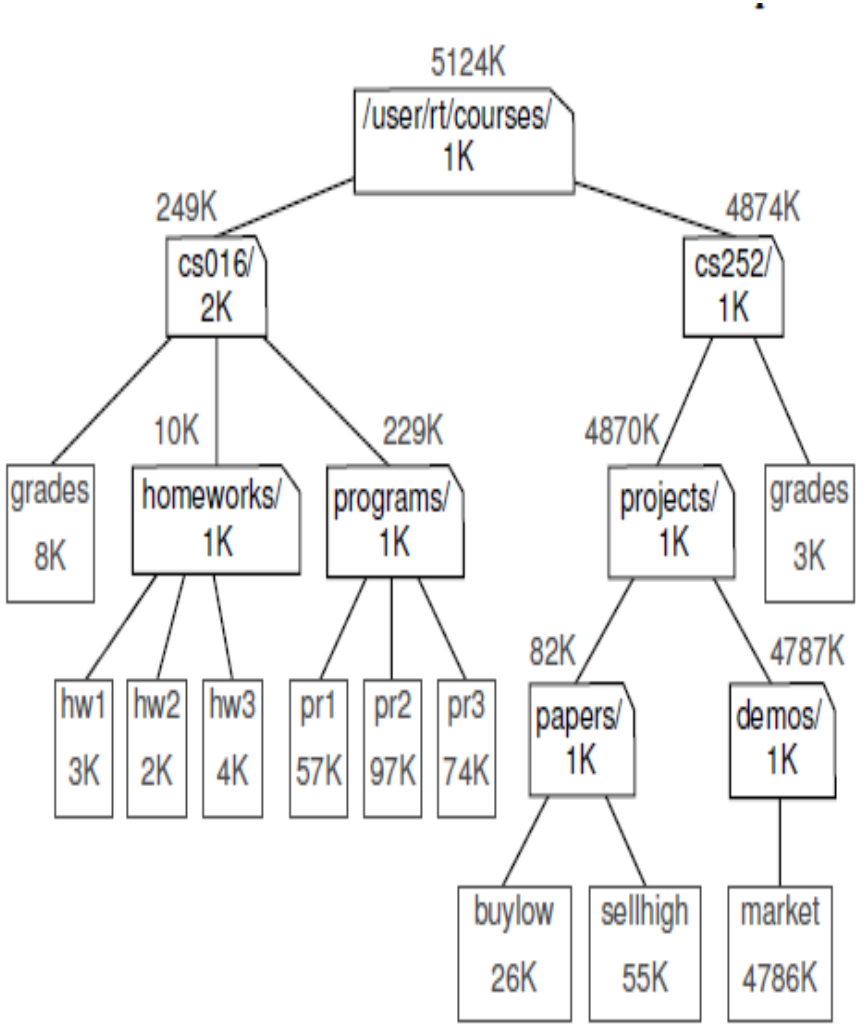
Recursão
Arquivos de Sistemas (cont.)

- Algumas operações tais como cópia ou deleção de um diretório são realizadas usando-se algoritmos recursivos



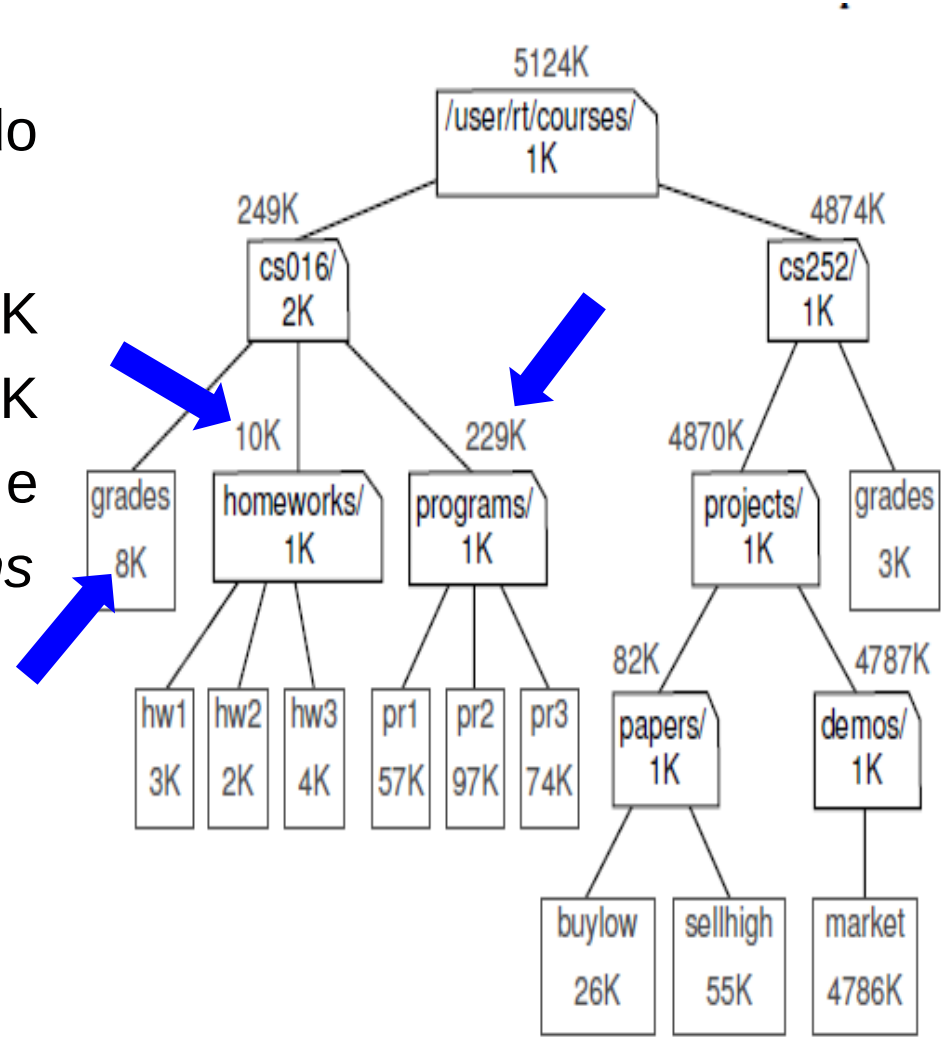
Recursão
Arquivos de Sistemas (cont.)

- O espaço em disco utilizado por uma entrada pode ser computado por meio de um simples algoritmo recursivo
  - O espaço utilizado é igual ao espaço utilizado pela entrada mais a soma do espaço acumulado das outras entradas que estão armazenadas diretamente dentro da entrada



Recursão
Arquivos de Sistemas (cont.)

- Exemplo:
  - O espaço em disco acumulado para o diretório *cs016* é 249K
  - *cs016* usa 2K e mais 8K acumulados de *grades*, 10K acumulados de *homeworks* e 229K acumulados de *programs*



Recursão
Arquivos de Sistemas (cont.)

- O pseudocódigo para o algoritmo de calculo de utilização de espaço é dado a seguir

```
Algorithm DiskUsage(path):
    Input: A string designating a path to a file-system entry
    Output: The cumulative disk space used by that entry and any nested entries
    total = size(path)                    {immediate disk space used by the entry}
    if path represents a directory then
        for each child entry stored within directory path do
            total = total + DiskUsage(child)          {recursive call}
    return total
```

Recursão
Arquivos de Sistemas (cont.)

- O código em Python para o algoritmo de calculo de utilização de espaço é dado a seguir

```
import os

def disk_usage(path):
  """Return the number of bytes used by a file/folder and any
descendents."""
  total = os.path.getsize(path)              # account for direct usage
  if os.path.isdir(path):                    # if this is a directory,
    for filename in os.listdir(path):           # then for each child:
      childpath = os.path.join(path, filename)# compose full path to child
      total += disk_usage(childpath)           # add child's usage to total

  print ('{0:<7}'.format(total), path)      # descriptive output (optional)
  return total
```

Recursão
Arquivos de Sistemas (cont.)

- O código em Python utiliza as seguintes funções de sistema

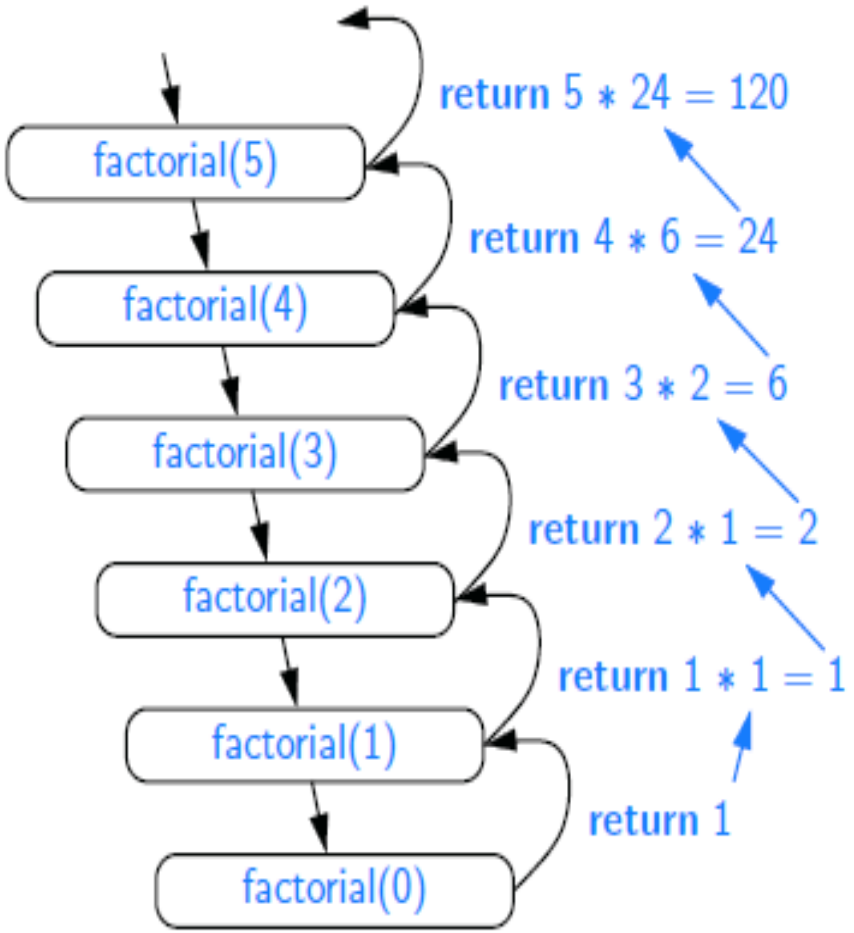
- **os.path.getsize(path)**
  Return the immediate disk usage (measured in bytes) for the file or directory that is identified by the string path (e.g., `/user/rt/courses`).
- **os.path.isdir(path)**
  Return True if entry designated by string path is a directory; False otherwise.
- **os.listdir(path)**
  Return a list of strings that are the names of all entries within a directory designated by string path. In our sample file system, if the parameter is `/user/rt/courses`, this returns the list `['cs016', 'cs252']`.
- **os.path.join(path, filename)**
  Compose the path string and filename string using an appropriate operating system separator between the two (e.g., the / character for a Unix/Linux system, and the \ character for Windows). Return the string that represents the full path to the file.

Recursão

Análise de Algoritmos Recursivos - Fatorial

- Para a função ***factorial*** (***n***) há um total de ***n*** + 1 ativações, com o parâmetro decrescendo de ***n*** na primeira chamada, para ***n*** - 1 na segunda chamada e assim por diante até alcançar o caso base com valor = 0
- Cada chamada da função executa um número constante de operações
- O número de operações para computar a função é ***O***(***n***), pois existem ***n*** + 1 chamadas, cada uma com ***O***(1) operações

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n-1)
```



Recursão
Análise de Algoritmos Recursivos – Busca Binária

- Considerando-se o tempo de execução do algoritmo de busca binária, observa-se que há um número constante de operações (primitivas) executadas em cada chamada recursiva do método
  - O tempo de execução é proporcional ao número de chamadas recursivas realizadas
  - Há no máximo $\lfloor \log n \rfloor + 1$ chamadas recursivas que são realizadas durante a busca binária de uma sequencia com ***n*** elementos
- Proposição:
  - O algoritmo de busca binária (*binary search algorithm*) tem um tempo de execução ***O***(log ***n***) para uma sequência ordenada com ***n*** elementos

Recursão
Análise de Algoritmos Recursivos – Busca Binária – (cont.)

- Justificativa da proposição:

Initially, the number of candidates is n; after the first call in a binary search, it is at most $n/2$; after the second call, it is at most $n/4$; and so on. In general, after the j^{th} call in a binary search, the number of candidate entries remaining is at most $n/2^j$. In the worst case (an unsuccessful search), the recursive calls stop when there are no more candidate entries. Hence, the maximum number of recursive calls performed, is the smallest integer r such that

$$\frac{n}{2^r} < 1.$$

In other words (recalling that we omit a logarithm's base when it is 2), $r > \log n$. Thus, we have

$$r = \lfloor \log n \rfloor + 1,$$

which implies that binary search runs in $O(\log n)$ time. ■

Recursão
Análise de Algoritmos Recursivos – Busca Binária – (cont.)

- Item not in the array (size N)
- T(N) = number of comparisons with array elements
- T(1) = 1
- T(N) = 1 + T(N / 2) ←
  = 1 + [1 + T(N / 4)]
  = 2 + T(N / 4) ←
  = 2 + [1 + T(N / 8)]
  = 3 + T(N / 8) ←
  = …
  = k + T(N / 2^k) [1]

Recursão
Análise de Algoritmos Recursivos – Busca Binária – (cont.)

- $T(N) = k + T(N / 2^k)$ [1]
- $T(N / 2^k)$ gets smaller until the base case: T(1)
  - $2^k = N$
  - $k = \log_2 N$
- Replace terms with k in [1]:

  $T(N) = \log_2 N + T(N / N)$

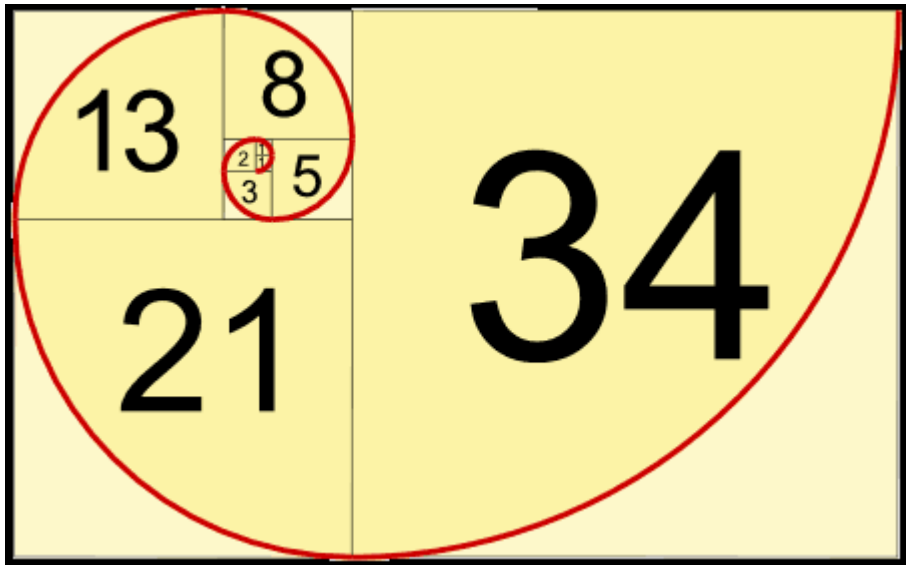
  $= \log_2 N + T(1)$

  $= \log_2 N + 1$
- $O(\log_2 N)$ algorithm
- We used *recurrence equations*

Recursão
Análise de Algoritmos Recursivos – Ineficiência na Recursão (1)

- A técnica de recursão é uma ferramenta muito poderosa, entretanto ela pode ser mal utilizada
- Seja a sequência de Fibonacci e a sua definição recursiva

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_n &= F_{n-2} + F_{n-1} \quad \text{for } n > 1.
\end{aligned}
$$



Recursão
Análise de Algoritmos Recursivos – Ineficiência na Recursão (cont.)

- Seja a implementação em Python baseada na definição dada

```
def bad_fibonacci(n):
  """Return the nth Fibonacci number."""
  if n <= 1:
    return n
  else:
    return bad_fibonacci(n-2) + bad_fibonacci(n-1)
```

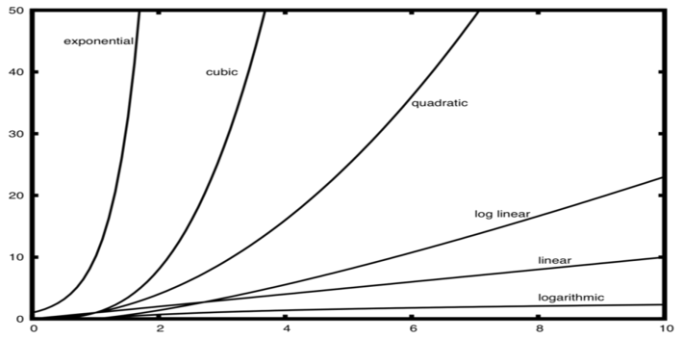
Recursão

Análise de Algoritmos Recursivos – Ineficiência na Recursão (cont.)

- A implementação proposta para o cálculo da sequência de Fibonacci é ineficiente
- Seja c_n = número de invocações da função de cálculo da sequência de Fibonacci para um número *n* qualquer. Então para o cálculo da sequência para *n* = 8, tem-se que

$$
\begin{aligned}
c_0 &= 1 \\
c_1 &= 1 \\
c_2 &= 1 + c_0 + c_1 = 1 + 1 + 1 = 3 \\
c_3 &= 1 + c_1 + c_2 = 1 + 1 + 3 = 5 \\
c_4 &= 1 + c_2 + c_3 = 1 + 3 + 5 = 9 \\
c_5 &= 1 + c_3 + c_4 = 1 + 5 + 9 = 15 \\
c_6 &= 1 + c_4 + c_5 = 1 + 9 + 15 = 25 \\
c_7 &= 1 + c_5 + c_6 = 1 + 15 + 25 = 41 \\
c_8 &= 1 + c_6 + c_7 = 1 + 25 + 41 = 67
\end{aligned}
$$

Para um *n* qualquer, tem-se então que o número de ativações é $c_n > 2^{n/2}$



Recursão
Análise de Algoritmos Recursivos – Ineficiência na Recursão (cont.)

- Seja a implementação em Python

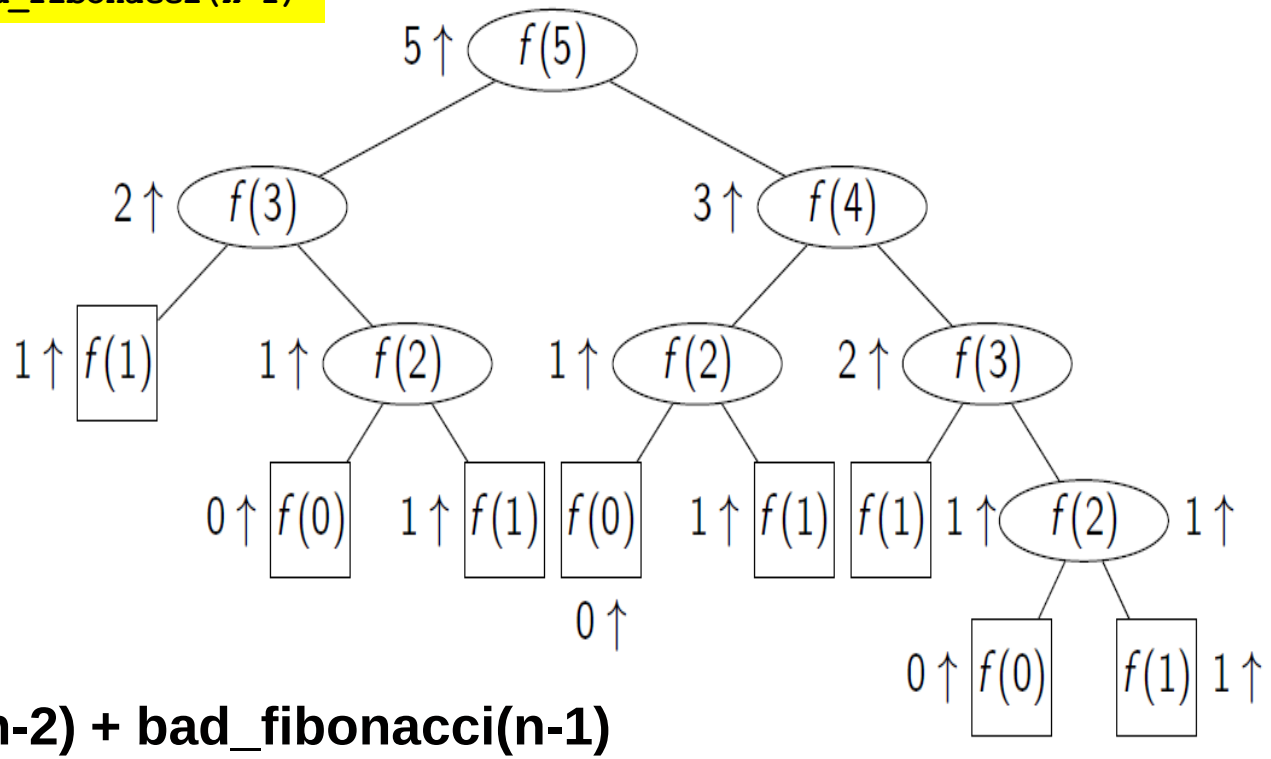
```
cont = 0
def bad_fibonacci(n):
  """Return the nth Fibonacci number."""
  global cont
  if n <= 1:
    cont = cont + 1
    print (cont)
    return n
  else:
    cont = cont + 1
    print (cont)
    return bad_fibonacci(n-2) + bad_fibonacci(n-1)
print (bad_fibonacci(8))
```

```
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
21
```

Recursão
Análise de Algoritmos Recursivos – Ineficiência na Recursão (cont.)

- Seja a implementação em Python baseada na definição dada

```
def bad_fibonacci(n):
  """Return the nth Fibonacci number."""
  if n <= 1:
    return n
  else:
    return bad_fibonacci(n-2) + bad_fibonacci(n-1)
```



return bad_fibonacci(n-2) + bad_fibonacci(n-1)

Recursão
Análise de Algoritmos Recursivos – Ineficiência na Recursão (cont.)

- Pode-se computar a função F_n de forma mais eficiente fazendo-se uma recursão que cada invocação faz somente uma chamada recursiva
  - Redefine-se também a função para retornar um par de números de Fibonacci consecutivos (F_n, F_{n-1}) e fazendo-se $F_{-1} = 0$
  - O tempo de execução é ***O*(*n*)**

```
def good_fibonacci(n):
  """Return pair of Fibonacci numbers, F(n) and F(n-1)."""
  if n <= 1:
    return (n,0)
  else:
    (a, b) = good_fibonacci(n-1)
    return (a+b, a)
```

Recursão

Análise de Algoritmos Recursivos – Ineficiência na Recursão (cont.)

```
cont = 0
def good_fibonacci(n):
  """Return pair of Fibonacci numbers, F(n) and F(n-1)."""
  global cont
  if n <= 1:
    cont = cont + 1
    print (cont)
    return (n,0)
  else:
    cont = cont + 1
    print (cont)
    (a, b) = good_fibonacci(n-1)
    return (a+b, a)
print (good_fibonacci(8))
```

```
========= RESTART: C:/Python-36/Codigo-Python-PEDS/good_fibonacci.py =========
1
2
3
4
5
6
7
8
(21, 13)
```

Recursão

Análise de Algoritmos Recursivos – Profundidade da Recursão no Python

- Outro perigo na utilização da recursão é a denominada recursão infinita
  - Se cada chamada recursiva faz outra chamada recursiva, sem nunca atingir o caso base, então tem-se uma série infinita de tais chamadas
  - Há um consumo exagerado de recursos
- Exemplo de recursão infinita

```
def fib(n):
    return fib(n) # fib(n) equals fib(n)
print (fib (1))
```

```
>>>
============== RESTART: D:/Python36/Teste/recursao_infinita.py
==============
Traceback (most recent call last):
  File "D:/Python36/Teste/recursao_infinita.py", line 3, in <module>
    print (fib (1))
  File "D:/Python36/Teste/recursao_infinita.py", line 2, in fib
    return fib(n) # fib(n) equals fib(n)
  File "D:/Python36/Teste/recursao_infinita.py", line 2, in fib
    return fib(n) # fib(n) equals fib(n)
  File "D:/Python36/Teste/recursao_infinita.py", line 2, in fib
    return fib(n) # fib(n) equals fib(n)
  [Previous line repeated 990 more times]
RecursionError: maximum recursion depth exceeded
>>>
```

Recursão
Análise de Algoritmos Recursivos – Profundidade da Recursão no Python

- A linguagem Python disponibiliza de um limite para brecar recursões infinitas
  - O limite padrão é de 1000 invocações
- Para o algoritmo de busca binária alcançar este limite, seria necessário 2^{1000} elementos
- Entretanto, caso seja necessário pode-se ultrapassar este limite

```
>>> import sys
>>> old = sys.getrecursionlimit( ) # perhaps 1000 is typical
>>> print (old)
1000
>>> sys.setrecursionlimit(1000000) # change to allow 1 million nested calls
>>> new = sys.getrecursionlimit( ) # perhaps 1000 is typical
>>> print (new)
1000000
```

Recursão
Análise de Algoritmos Recursivos – Classificação da Recursões

- Pode-se classificar as chamadas recursivas em função da quantidade de chamadas da seguinte forma:
  - Recursão linear, onde no máximo uma chamada é realizada
  - Recursão binária, onde podem ocorrer duas chamadas recursivas
  - Recursão múltipla, onde podem ser realizadas mais de duas chamadas recursivas

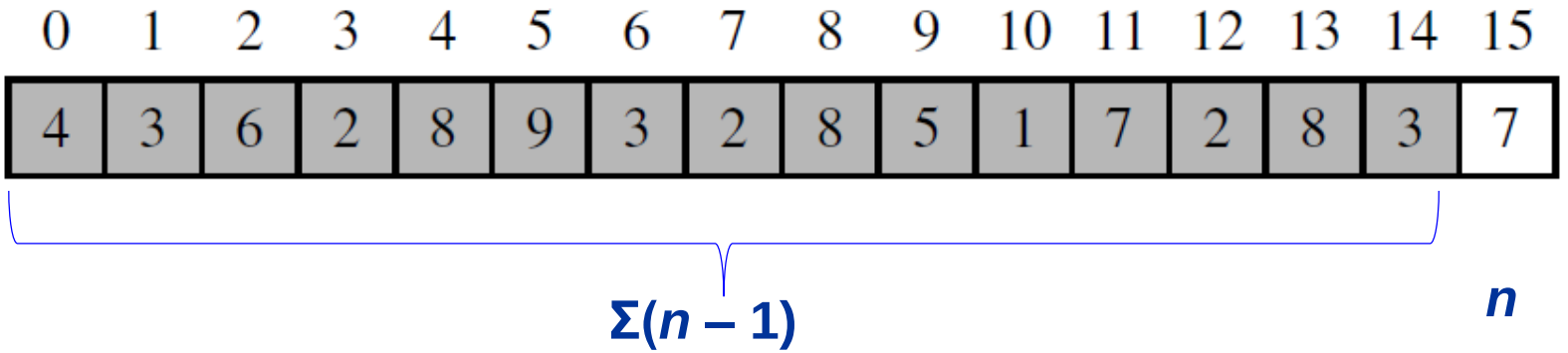
Recursão
Análise de Algoritmos Recursivos – Recursão Linear

- Se uma função recursiva é projetada para que cada invocação dela seja no máximo uma nova chamada recursiva, isso é conhecido como recursão linear
- Exemplos
  - factorial
  - good_fibonacci
  - binary_search

```
1  def factorial(n):
2    if n == 0:
3      return 1
4    else:
5      return n * factorial(n-1)
```

Recursão
Análise de Algoritmos Recursivos – Soma de Elementos

- A recursão linear pode um instrumento útil para o processamento de uma sequência de dados, como uma lista Python
- Suponha que se quer computar a soma de uma sequência ***S*** de ***n*** inteiros. Pode-se resolver este problema de somatório utilizando-se de recursão linear
  - Observa-se que a soma de todos os ***n*** inteiros em ***S*** é 0 se ***n*** = 0, caso contrário a soma é constituída da soma dos primeiros ***n*** – 1 inteiros mais o último elemento em ***S***



Recursão
Análise de Algoritmos Recursivos – Soma de Elementos (cont.)

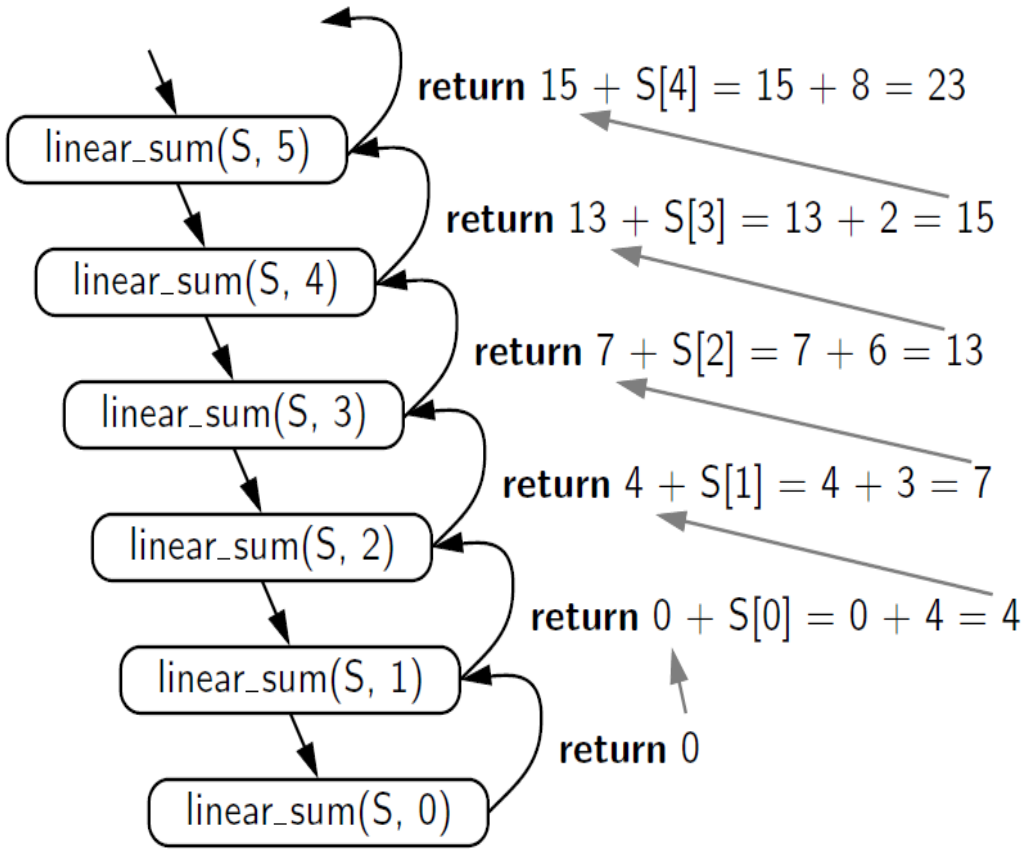
- A implementação do algoritmo é a seguinte

```
def linear_sum(S, n):
  """Return the sum of the first n numbers of sequence S."""
  if n == 0:
    return 0
  else:
    return linear_sum(S, n-1) + S[n-1]
```

Recursão
Análise de Algoritmos Recursivos – Soma de Elementos (cont.)

- A implementação do algoritmo é a seguinte

```
def linear_sum(S, n):
  """Return the sum of the first n numbers of sequence S."""
  if n == 0:
    return 0
  else:
    return linear_sum(S, n-1) + S[n-1]
```

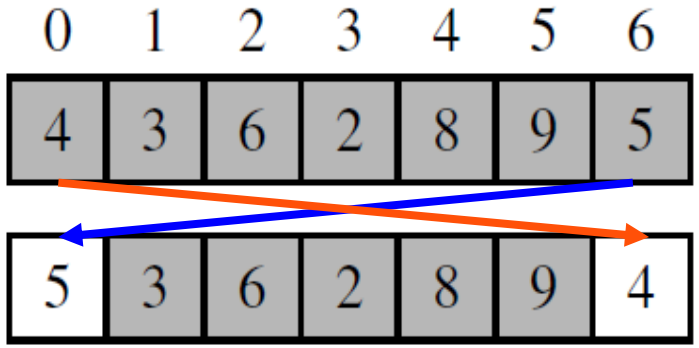


O tempo de execução é ***O*(*n*)**
O uso de memória é ***O*(*n*)**

Recursão

Análise de Algoritmos Recursivos – Inversão de uma Lista

- Outro problema a ser considerado é a inversão de uma lista ***S*** de ***n*** elementos, de modo que o primeiro elemento se transforme no último, o segundo se transforme no penúltimo e assim por diante
- Pode-se resolver este problema utilizando-se recursão linear
- Observa-se que a inversão de uma lista pode ser obtida fazendo-se a troca (*swapping*) do primeiro e último elemento e então invertendo recursivamente os outros elementos da lista



Recursão
Análise de Algoritmos Recursivos – Soma de Elementos (cont.)

- A implementação do algoritmo é a seguinte

```
def reverse(S, start, stop):
  """Reverse elements in implicit slice S[start:stop]."""
  if start < stop - 1:                              # if at least 2 elements:
     S[start], S[stop-1] = S[stop-1], S[start]  # swap first and last
     reverse(S, start+1, stop-1)                  # recur on rest
```

Recursão
Análise de Algoritmos Recursivos – Soma de Elementos (cont.)

- A implementação do algoritmo é a seguinte

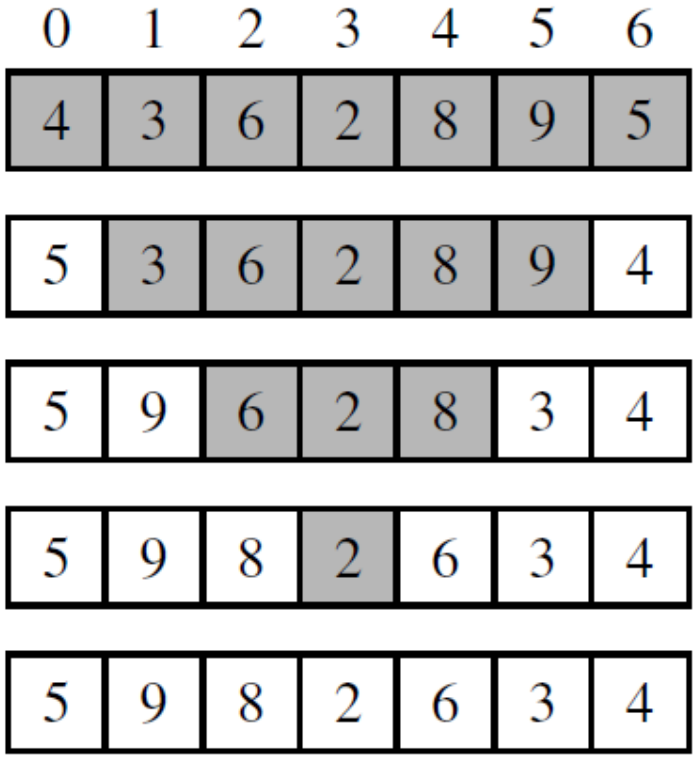
```
def reverse(S, start, stop):
  """Reverse elements in implicit slice S[start:stop]."""
  if start < stop - 1:                          # if at least 2 elements:
      S[start], S[stop-1] = S[stop-1], S[start] # swap first and last
      reverse(S, start+1, stop-1)               # recur on rest
```

Há dois casos bases:

1) *start* == *stop* (o intervalo está vazio)

2) *start* == *stop* -1 (só há um elemento)

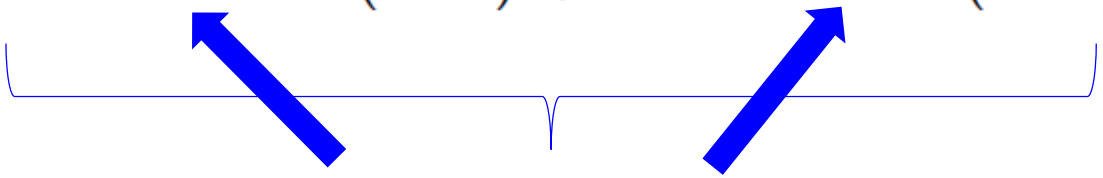
Na recursão garante-se fazer progresso em direção ao caso base, com a diferença *stop* - *start* diminuindo por 2 em cada chamada



Recursão
Análise de Algoritmos Recursivos – Recursão Binária

- Quando uma função faz duas chamadas recursivas, diz-se que ela utiliza recursão binária

```
1  def bad_fibonacci(n):
2    """Return the nth Fibonacci number."""
3    if n <= 1:
4      return n
5    else:
6      return bad_fibonacci(n-2) + bad_fibonacci(n-1)
```



2 chamadas recursivas

Recursão
Análise de Algoritmos Recursivos – Recursão Binária

- O problema da soma de ***n*** elementos de uma lista ***S*** pode ser resolvido utilizando-se de recursão binária
  - Computar-se a soma de zero ou um elementos é trivial
  - Para se computar a soma de dois ou mais elementos, pode-se recursivamente computar recursivamente a soma da primeira metade dos elementos, a soma da segunda metade e somar as somas obtidas

Recursão
Análise de Algoritmos Recursivos – Recursão Binária

- A implementação do algoritmo em Python é a seguinte

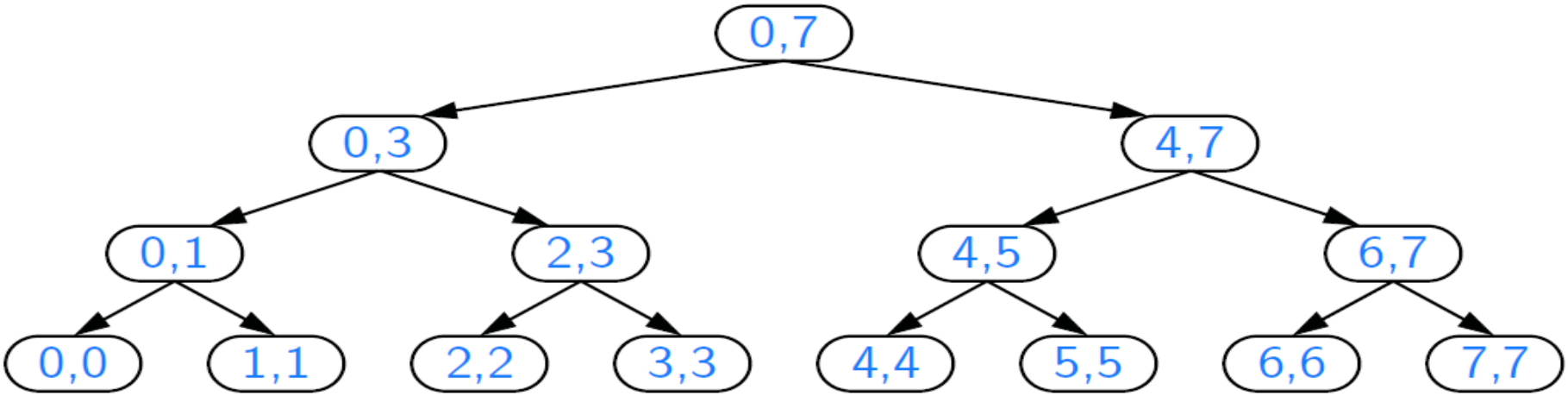
```
def binary_sum(S, start, stop):
  """Return the sum of the numbers in implicit slice S[start:stop]."""
  if start >= stop:                    # zero elements in slice
    return 0
  elif start == stop-1:                # one element in slice
    return S[start]
  else:                                # two or more elements in slice
    mid = (start + stop) // 2
    return binary_sum(S, start, mid) + binary_sum(S, mid, stop)
```

Recursão
Análise de Algoritmos Recursivos – Recursão Binária

- A implementação do algoritmo em Python é a seguinte

```
def binary_sum(S, start, stop):
  """Return the sum of the numbers in implicit slice
S[start:stop]."""
  if start >= stop:                  # zero elements in slice
    return 0
  elif start == stop-1:              # one element in slice
    return S[start]
  else:                              # two or more elements in slice
    mid = (start + stop) // 2
    return binary_sum(S, start, mid) + binary_sum(S, mid, stop)
```

O tempo de execução é ***O***(***n***)
O uso de memória é ***O***(log***n***)



Recursion trace for the execution of binarySum(data, 0, 7).

Recursão
Análise de Algoritmos Recursivos – Recursão de Cauda

- O principal benefício de uma abordagem recursiva no projeto de algoritmos é que ela permite tirar proveito de uma estrutura repetitiva presente em muitos problemas.
  - Fazendo-se o algoritmo explorar a estrutura repetitiva de uma maneira recursiva, muitas vezes pode-se evitar a análise de casos complexos e laços aninhados.
  - Essa abordagem pode levar a descrições de algoritmos mais legíveis, e ainda bastante eficientes.
- No entanto, a utilidade da recursão tem um custo
  - O interpretador Python deve manter registros de ativação que controlem o estado de cada chamada recursiva. Quando a memória é um recurso altamente escasso, pode ser útil em alguns casos poder derivar algoritmos não recursivos de algoritmos recursivos

Recursão
Análise de Algoritmos Recursivos – Recursão de Cauda

- Uma recursão é dita ser recursão de cauda (*tail recursion*) se qualquer chamada recursiva que seja feita no contexto é a última operação deste contexto, com o valor da chamada recursiva sendo retornado imediatamente
- A recursão de cauda necessariamente só ocorre com recursão linear

```
1  def reverse(S, start, stop):
2    """Reverse elements in implicit slice S[start:stop]."""
3    if start < stop - 1:                    # if at least 2 elements:
4      S[start], S[stop-1] = S[stop-1], S[start]    # swap first and last
5      reverse(S, start+1, stop-1)                  # recur on rest
```

recursão de cauda 

Recursão
Análise de Algoritmos Recursivos – Recursão de Cauda (cont.)

```
1  def binary_search(data, target, low, high):
2    """Return True if target is found in indicated portion of a Python list.
3
4    The search only considers the portion from data[low] to data[high] inclusive.
5    """
6    if low > high:
7      return False                          # interval is empty; no match
8    else:
9      mid = (low + high) // 2
10     if target == data[mid]:               # found a match
11       return True
12     elif target < data[mid]:
13       # recur on the portion left of the middle
14       return binary_search(data, target, low, mid - 1)
15     else:
16       # recur on the portion right of the middle
         return binary_search(data, target, mid + 1, high)
```

```
1  def factorial(n):
2    if n == 0:
3      return 1
4    else:
5      return n * factorial(n-1)
```

Não é uma recursão de cauda

Recursão
Análise de Algoritmos Recursivos – Recursão de Cauda (cont.)

- Qualquer recursão de cauda pode ser reimplementada de forma não recursiva, como o caso da função *binary_search_iterative*, versão iterativa da função *binary_search*

```
def binary_search_iterative(data, target):
  """Return True if target is found in the given Python list."""
  low = 0
  high = len(data)-1
  while low <= high:
    mid = (low + high) // 2
    if target == data[mid]:        # found a match
      return True
    elif target < data[mid]:
      high = mid - 1               # only consider values left of mid
    else:
      low = mid + 1                # only consider values right of mid
  return False                     # loop ended without success
```